A Software Design Approach for Heterogeneous Systems of Unattended Sensors, Unmanned Vehicles and Monitoring Stations

William J. Smuda*[a], Grant Gerhart [a], Man-Tak Shing [b], Mikhail Auguston [b]
[a]US Army, Tank Automotive RDE Center, 6501 E. 11 Mile Rd. Warren, MI, USA, 48397-5000;
[b]Naval Postgraduate School, Monterey, CA 93943

ABSTRACT

The design and implementation of software for network systems of diverse physical assets is a continuing challenge to sensor network developers. The problems are often multiplied when adding new elements, and when reconfiguring existing systems. For software systems, like physical systems, explicit architectural descriptions increase system level comprehension. Coupled with well defined object oriented design practices, system extensibility is defined and software reuse and code composition are enabled.

Our research is based on model driven design architecture. High level system models are defined in the Unified Modeling Language (UML), the language of the software engineer. However, since most experimental work is done by non-software specialists, (electronics Engineers, Mechanical Engineers and technicians) the model is translated into a graphical, domain specific model. Components are presented as domain specific icons, and constraints from the UML model are propagated into the domain model. Domain specialists manipulate the domain model, which then composes software elements needed at each node to create an aggregate system.

Keywords: Software Design, Unattended Sensors, Unmanned Vehicles, Model Driven Design, Software Reuse, UML, Distributed Systems, Design Patterns, Software Components, JAUS

1. INTRODUCTION

1.1. Military Robotics and Unattended Sensors

Unattended Sensor and Unmanned ground vehicle (UGV) technology can be used in a number of ways to assist in counter-terrorism activities now. Unattended sensors have wide application in surveillance and perimeter monitoring. In addition to the conventional uses of tele-operated robots for unexploded ordinance handling and disposal, water cannons and other crowd control devices, robots can also be employed for a host of terrorism deterrence and detection applications. Due to the immaturity of sensors and intelligent algorithms, we have found that as recently as 3 years ago, users were not ready for fully autonomous vehicles [1]. The same was true of autonomous sensor networks. However, as we move to the future, with the wider deployment of unattended sensors and robotics, as well as the emergence of new sensors and algorithms, requests for autonomy are already being heard. We still hold to the tenant that autonomous behavior is complexly intertwined with autonomous mission understanding, Figure 1. It does no good to send an autonomous vehicle into the danger zone unless we are sure that an event will be detected and noted.

Human performance studies were conducted by the US Army Research Institute to explore new approaches for battle command as may be experienced by soldiers using the Future Combat System (FCS). FCS concepts call for unprecedented integration of automation, sensors and robotics. One of the FCS goals is to reduce the size of the command group. The challenge is to find the optimum workload for command group soldiers. As expected, as workload increases, at the "too-high" levels of complexity, the information and battle space managers' performance drops sharply [2]. Our challenge is too invent fused sensor information and mission awareness tools to reduce the amount of information that the human in the loop needs to process and communicate to their associates.

Sensor Fusion, Mission Planning and Mission Awareness are usually associated with autonomous operation, but can also apply to mission package data. In either case, the goal is to provide some hardware/software module to reduce the data load on the operator and/or enable automation of robotic operation [3]. Our first goal is to remove personnel from the danger zone. In automotive applications we can sometimes create more sensitive sensors that alert the operator to a hazard with time for human reaction. For military and police activities, this is often technically unfeasible or cost prohibitive; a solution is to move sensors into the danger zone on a robotic mobility platform. In either case, the key is creating modules to interpret sensor data and alert the human operator that a hazard is near.

Report Documentation Page		*Form Approved* *OMB No. 0704-0188*

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **08 MAR 2004**	2. REPORT TYPE **Journal Article**	3. DATES COVERED **08-03-2004 to 08-03-2004**
4. TITLE AND SUBTITLE **A Software Design Approach for Heterogeneous Systems ofUnattended Sensors, Unmanned Vehicles and Monitoring Stations**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **William Smuda; Grant Gerhart; Man-Tak Shing; Mikhail Auguston**		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,1 University Way,Monterey,CA,93943**		8. PERFORMING ORGANIZATION REPORT NUMBER **; #16019**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) **U.S. Army TARDEC, 6501 East Eleven Mile Rd, Warren, Mi, 48397-5000**		10. SPONSOR/MONITOR’S ACRONYM(S) **TARDEC**
		11. SPONSOR/MONITOR’S REPORT NUMBER(S) **#16019**

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
SPIE Europe Defense & Security Symposium 2006

14. ABSTRACT
The news release of the Multi Services Electro-optics Signature(MuSES) code includes the ability to customize almost every simulation aspect. User routines enable simulation of complex phenomena using the existing thermal solver. By provding easy to use methods to customize virtually every input, fast and accurate solutions are calculated. This customization is further advanced with the addition of hook functions and the Application Programming Interface(API). Hook functions allow the user to have solver level interactions without adding complexity, while the API allows the user to retrieve and override values within the solver. Together, user routines, hook functions, and the API give the user the capability to model advanced heating algorthms, complex control and logic systems, and proprietary survivability techniques without considerable learning curve.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Public Release**	**17**	

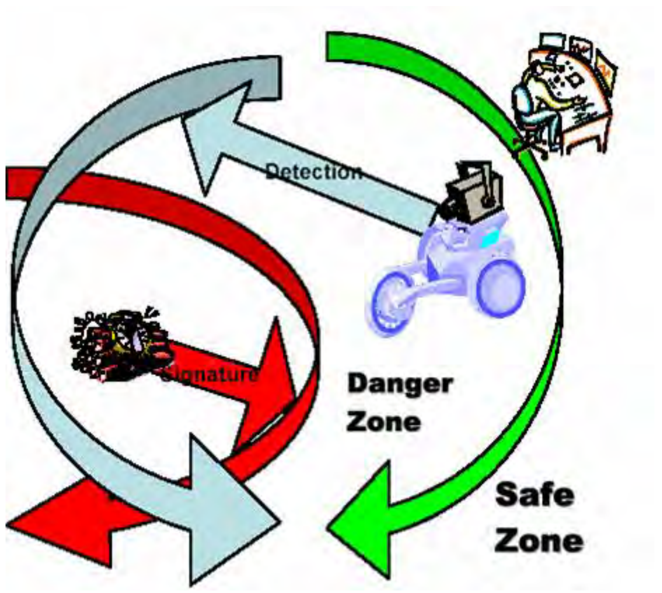

Figure 1 The Effective Area – Overlap Between Signature Horizon and Detection Horizon.

Integrating these modules is a software intensive task in most cases. To be responsive to user need, we must have tools and architectures in place to rapidly integrate sensors, mission planning and mission awareness modules as they mature.

1.2. Design Assumptions

We begin with design assumptions based on experiences gained over the last several years.

The first assumption is that we have a collection of artifacts that we are interested in integrating. These artifacts may include Operator Control Units, Platforms (robots or unattended sensors), mission sensors, proprioceptive sensors control algorithms or mission packages (arms, masts etc.) just to name a few. In most cases, these disparate artifacts do not conform to messaging standards. In most cases, we do not have access to the embedded processors to include additional code. In most cases, we do not have access to the code, or access to proprietary compilers needed to modify the code. In essence, we want to integrate a collection of black boxes. We have some knowledge of the physical I/O, but the software data structures needed to communicate must often be ferreted out from code or by inspection of the run-time communications. In the worst case, all integration software will run on auxiliary processors.

Needless to say, creating interfaces to these artifacts can be an expensive and time consuming effort; effort we would like to retain and reuse. We would also like to make this information and knowledge usable by non-software experts.

The second assumption is that in most government labs, software engineers are a scarce commodity. That is not to say we don't have software "guys". We have them and many are talented individuals, however, they are often not trained in the intricacies of modern software design and development paradigms.

In summary, the worst and often typical case is that we need to integrate a collection of artifacts that we can only access via external interfaces. The engineers and scientists are usually robotic or unmanned sensor specialists with a smattering of software knowledge. Experienced software engineers and experienced robotic and unattended sensor engineers with intensive software engineering experience are in short supply.

Our task then is to develop guidelines, methods and tools to:

- Capture Software Engineering Expertise.
- Transfer this knowledge to Domain Engineers.
- Capture software elements for reuse.
- Capture configuration and execution data.
- Provide tools to simplify the integration process.

2. DESIGN ELEMENTS

2.1. Top Level Design

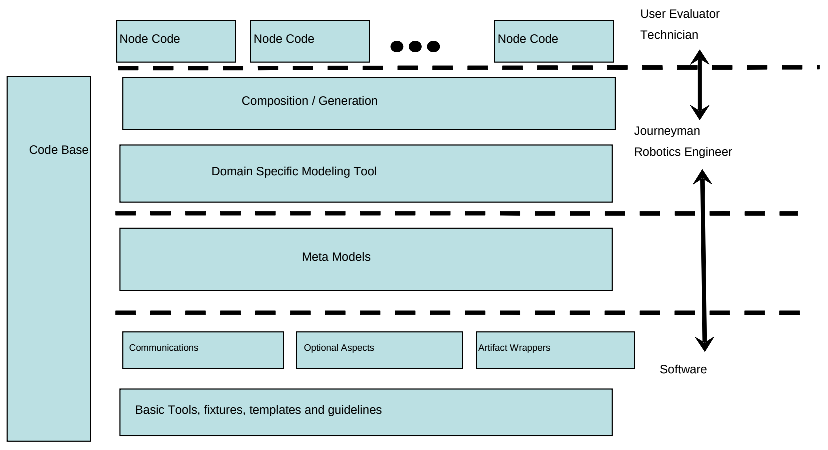

Figure 2 Top Level Design

Referring to Figure 2, we find a PowerPoint representation of the proposed system. As you can see, it is a simple block diagram of high level abstractions. The placement of the abstractions indicates a progression from flexible sub architecture, the realm of the software engineer, to a generic sub-architecture the realm of the robotic and/or unattended sensor engineer, to a product, the realm of the user; a technical evaluator of a robotic system in this case.

2.1.1. Code Base

Staring on the left, there is a block labeled “Code Base”. This block represents a storage abstraction. It can be as formal as the Database, or it can be a simple set of files in a folder. As the project matures, we expect the former, but there may be some new storage paradigm that may be implemented in the future.

2.1.2. Use Cases

On the right, there are simple arrows, showing the range of human actors in the system. There are no exact limitations, but we notice a progression of responsibility from the software engineer at the bottom to the technician at the top.

2.1.3. Foundation

The center bottom block is labeled Basic Tools, fixtures and guidelines. These may include editors, XML parsers, design patterns and text files describing requirements and standards. This abstraction represents a foundation for what is to come.

2.1.4. Components

The next set of blocks up represents reusable components. These components are stored in the code base when completed. The final codes are composed from this set of components. The components are constructed using design patterns described in the bottom block. Different types of components will use different design patterns. The design patterns are necessary to insure that the interfaces are of the proper type at composition time. Notice that there is a dashed line above the component blocks. This line indicates that this part of the architecture is not only in the realm of the software engineer, but also indicates a temporal abstraction. The guidelines and components are a necessary prerequisite to the blocks above. This relationship is not a hard one. In order for the meta model to work, it needs to know at least, what components will be available to the final composition. Additional components may be added as time goes on, but new meta models will need to be assembled to take advantage of them.

2.1.5. Meta Model

The third block up is titled Meta Models. This is the spine of the architecture. Meta models are created by software engineers with knowledge of the domain or software engineers collaborating with domain experts. They are the key to this architecture. The meta model encapsulates high level information about the system. The meta model defines component relation ships and constraints. It is a vehicle to encapsulate software engineering knowledge and facilitate transfer of this knowledge to domain experts.

Above the meta model is another dashed line; another separation of responsibility and another temporal relationship. Blocks above the meta model cannot be realized until the meta model is complete.

2.1.6. Domain Model

The fourth block up in the architecture is the Domain specific modeling tool. This tool is generated from the meta model. It is a workspace from which concrete models of the system under construction may be instantiated.

There may be several or many domain models created from a single meta model. The domain model is often constructed with icons specific to the domain being examined. This enables domain engineers to create system models for a variety of scenarios. For instance, the meta model may include a communications element. The domain modeler may choose from a variety of concrete communications components like serial, TCP/IP or CAN communication components. The domain modeler is also able to select the individual end nodes that will participate in the system.

2.1.7. Putting it all Together

Moving up to the fifth level, Composition / Generation allows the domain expert to create the software elements necessary, again without having to know the software engineering details necessary to accomplish this task. Again, the goal is to separate concerns. This architecture allows the experts to do what they know. Software engineers do not need to learn intricate details of the realization of the system; domain engineers do not need to know the details of the software engineering needed to provide them with this tool. “Architects” create models of buildings, structural engineers flesh out the design depending on location, customer and environmental factors. Tradesmen build the building.

Above the fifth block is yet another dashed line; another separation of responsibility and another temporal break point. Transitioning across this line is not possible until all the blocks below have been realized. It also is the transition from design to a reification of the system. Above the line is the artifact of interest, something that can be used for experimentation or as a production item.

2.1.8. Nodes

The top set of blocks is the node code. Once the codes are generated, they can be move to the individual nodes. The aggregate is the system artifact. In a prototyping environment, the artifact can be run through its paces in a variety of mission scenarios. The final artifact may be a simulation, a hardware-in-the-loop simulation or a pre-production hardware prototype. At the very top, there are blocks labeled simply node code. These blocks are an abstraction of a final product. In the case of a robotic prototyping system, each node has a run time architecture associated with it, generated from the domain model, which is an instance of the meta model (Figure 3). The run time system accepts XML messages, operates on the messages to modify behavior (such as throttling throughput) or to log events. It also parses the XML and converts to a format acceptable to the artifact of interest.

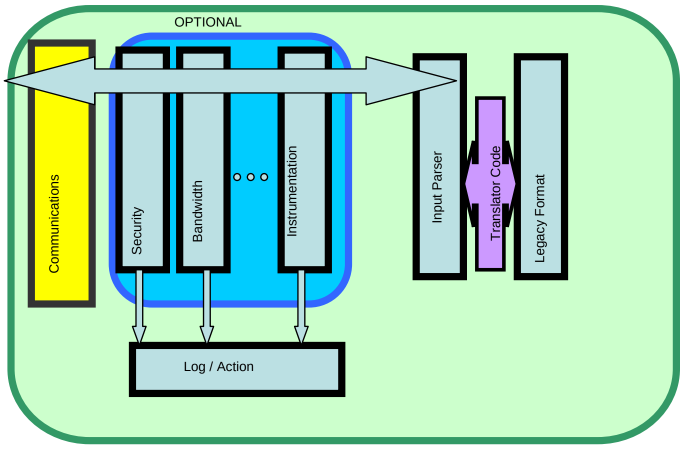

Figure 3 Prototyping System Node Run-Time Architecture

2.2. Model Driven Engineering

This design is based on Model Driven Engineering, a relatively new software development paradigm. Product line development for distributed embedded systems, such as aerospace and automotive, has become extremely complex. Developers spend years mastering platform APIs ands usage, even still, they often only come to a complete understanding of a subset of the platforms they develop for regularly. Model driven engineering focuses on abstractions particular to the application problem space and expresses designs in terms of concepts from that space. [5]

Model Driven Engineering combines software components constructed to conform to specific design patterns with Domain Specific Languages. These languages are described in a meta model, often graphical, that defines the relation ships of abstractions in the domain. The meta models are created in UML, the language of the software engineer. They are transformed into a constrained design environment, usually using graphical icons that pictorially describe the abstractions in terms easily understandable by domain engineers.

The domain engineers then create concrete instances of the meta model using icons that represent components available for composition of the final product. From the completed design, program generators are able to assemble components and create glue code to allow them to work together.

2.3. Standards & Tools

In many research applications, particularly in the early phases, standards often take a back seat. Engineers are encouraged to think outside the box or standards do not yet exist. However for this effort, several standards are of the up most importance.

XML standards are important at several levels. XML representations of models facilitate transitions between different phases of development and allow the use of automated tools.

The Joint Architecture for Unmanned Systems (JAUS) [4], transitioning to an SAE standard, provides a common messaging framework.

2.4. Components

Components are the key to software reuse. A collection of components is created by software engineers. These components later become a selection of model elements selectable by domain engineers.

Clemens Szyperski of Microsoft Corporation writes: “All components exist in a flat universe. This is an important property, as it allows servicing of components without having to know all places where that component has been used” [6]. This indicates that components should support a consistent interface and contract.

Both Szyperski, in Component Software [7] and Czarnecki and Eisenecker, in Generative Programming: Beyond Object Oriented Programming [8] agree that a component:

- Is a unit of independent deployment.
- Has no externally observable state.

However, Szyperski contends that a component is a unit of third party composition, while Czarnecki and Eisenecker relax the requirement of “third party composition”. We agree with Czarnecki and Eisenecker as long as the first two requirements hold. Components may be created internally or externally. Components are simple building blocks combinable in as many ways as possible.

For the purposes of this design, there are three general classes of components:

1. The endpoints, the individual hardware artifacts or simulation artifacts, along with wrappers, software, that at a minimum provides a mechanism to allow artifacts to be connected are components for our purposes.
2. An arbitrary number of optional components to instrument the prototype, induce disturbances, simulate communications protocols, throttle communications speed and/or provide translations to name a few.
3. Communications components that connect the nodes to the system. These may be very simple components such as TCP/IP or serial connection code. Or they may be very complex communication components such as

self organizing mesh networks, TCP/IP networks with additional discovery algorithms or entirely new communications components.

Each of the classes should conform to a common interface to allow automatic construction of the resulting run time code.

2.5. Design Patterns

Design patterns are high level abstractions of common design problems. They help us describe components, or collections of components. By using design patterns, we can develop designs that are extendable and mutable. If we create designs that specify a particular design pattern, we can take advantage of polymorphism and create new behaviors within this pattern at a later date and reuse the high level design. This means we can add new artifacts, optional components or communication components as needed.

To use design patterns effectively, we take advantage of common abstract interfaces. The glue code generators defined at the design level bring together a collection of interfaces. The details of the actual implementation below the interface are unimportant to the glue code generator. As an analogy, think of a soda bottling plant. The design for the plant includes a capping machine. The capping machine is concerned with the interface; the bottle top and the cap. It is not concerned with the flavor of the beverage inside the bottle. If the plant design is modified to change from a crimped cap to a screw cap, all the designers need to be concerned with is the interface, they do not need to be concerned with the flavor of the beverage that is being contained.

Design patterns might also be compared to composite digital devices. When creating an electronic design, we refer to reference material of discrete components. These components have a well defined interface. We may be concerned with some of the characteristics of these components dictated by their internal makeup, such as power consumption or latency, but we are not usually concerned about the intricate details. What we are concerned about is the interface. In order to compose a circuit, we need to know the pin outs and function of the device. We find this in a reference volume or specification sheet from the manufacture. In many cases, there may be more than one manufacture; the internals of the chip may be different, but the interface is common. This allows us to use tools that can layout the traces on a circuit board.

Design patterns are becoming a similar abstraction for software. A particular design pattern specifies an interface and function that the high level designer is interested in. Component developers do not need to know in what context the design pattern is being used; they need to know the function and interface they are creating.

Reference material is becoming available for software design patterns, just as there are reference volumes for electronics. There are several excellent books available for understanding design patterns:

> “Design Patterns, Elements of Reusable Object-Oriented Software” [9] provides a catalog of common general purpose patterns. “Head First Design Patterns” [10] is very readable introduction to the most common design patterns. It provides detailed examples with UML descriptions and Java code. “Pattern-Oriented Software Architecture, Vol 2, Patterns for Concurrent and Networked Objects” [11] provides patterns to solve the often difficult problems associated with communications in distributed systems.

Three main design patterns will be used in this work:

> **Adaptor.** The adaptor patterns will be used to wrap the legacy and research artifacts that represent the physical and control nodes of the robotic system. The input and outputs of the adaptor pattern will be XML representations of JAUS messages. The JAUS messages will be converted to the software and physical formats necessary to the artifact. As an example, the ODIS-T2 robot accepts proprietary data packets via a serial port. The wrapper will convert to and from JAUS message format to ODIS-T2 format; it will also transport the proprietary data packets via a serial link.

> **Visitor.** The visitor patterns will be responsible for passing the JAUS message to any instrumentation, modification or other optional component specified in the design. Visitors will insure that each incoming and outgoing message is seen and/or operated on by the optional components.

> **Proxy.** The communications components will be accessed via a proxy pattern. The communications components are expected to vary widely, from simple serial to very complex mesh networks with discovery. Using a proxy pattern insures that the system will be easily extensible. As the communications environment varies, only the proxy

design pattern will need to be modified. To the rest of the node, communications will be simply, a message has been received, or a message is being sent.

Additional design patterns may be used in conjunction with the main design patterns within components. This will simplify modifying and expanding components as the system matures.

2.6. Meta Model

The meta modeling environment for this project is the Generic Modeling Environment (GME) [12], an open source, visual, configurable environment for creating Domain Specific Modeling languages. GME use starts with configuration of the modeling environment; modeling of the modeling process or creating a meta model. The modeling language is UML class diagrams. Figure 4 shows a simple meta model for a robot system, the work under discussion.

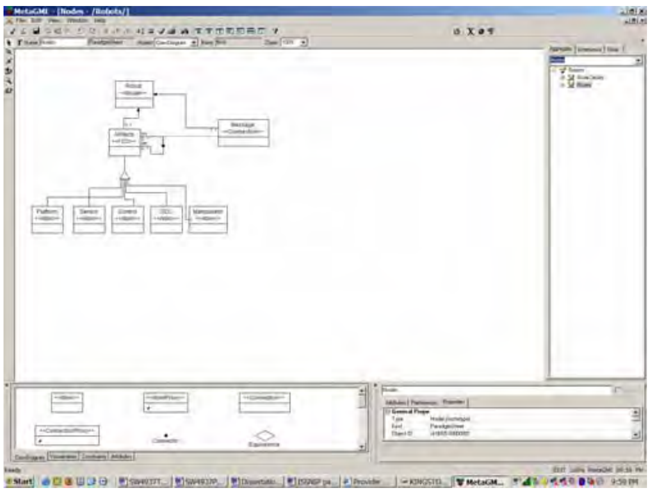

Figure 4 Simplified Meta Model of a Robotic System.

The meta model is a source document. That is, unlike previous CASE models of the 90's, it is not left behind to get out of sync with the implementation. The meta model defines a paradigm, a set of rules that will configure the GME for a specific operation.

In the case of Figure 4, the top level object is a model labeled "Robot". Contained in the Robot meta model are messages and artifacts. Artifacts are abstract; they do not have any implementation. The artifacts are defined by inherited types, bottom level objects or atoms. There are five different types of atoms possible to represent artifacts. A robot model can contain 1 or more artifacts. Finally connections between the artifacts are defined as "messages". Artifacts can send or receive 0 or more different messages.

If we wanted to configure, the final artifacts (and we do) they would be redefined from atoms to another type that allows containment. There may be multiple objects contained in each artifact, these in turn would be defined as atoms. Some of the Lower level objects we are interested in are wrappers for legacy or other non-conforming physical objects, instrumentation and communications components.

2.7. Domain Specific Model Language

The Domain Specific Modeling Language (DSML) is generated from a corresponding meta model. Note that due to configuration of the meta model, the artifacts are now represented by domain specific icons that represent their functionality in terms of the domain of interest; in this case, robots.

The new modeling environment is handed off to a domain engineer. The domain engineer selects from a palate of approved abstract artifacts, (controls, sensors, platforms, Operator Control Units (OCUs) and manipulators) and creates a model by connecting them in a meaningful way (Figure 5). The underlying components that will be uses are aspects of the artifacts dragged onto the work space.

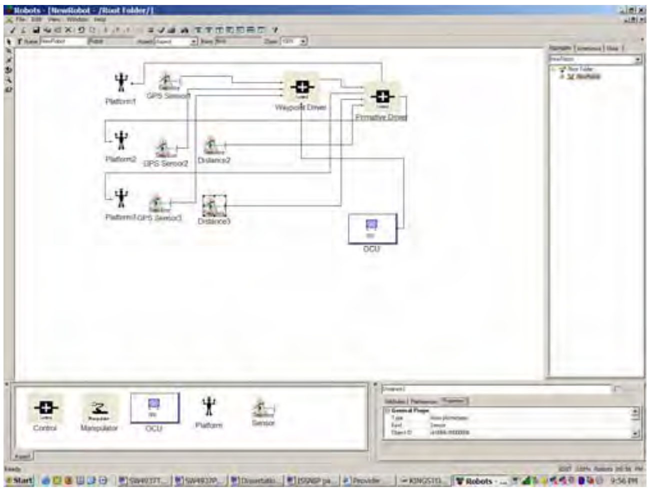

Figure 5 Domain Specific Modeling Workspace

There may be any number of models created by the domain engineers. The number of models is limited only by the cardinalities imposed in the meta model and the number of component instances available for each artifact. This particular model is of three robots, a leader and two followers. Each has a GPS positioning sensor and the two followers have distance sensors. The waypoint driver control computes waypoints for the two followers based on the input from the five sensors. The waypoint driver passes new messages to the primitive driver to control the two follower robots. The way point driver receives messages from the OCU to control the leader. The way point driver may be influenced by the messages from the OCU to vary distance for instance.

The meta model to model translation facilitates transfer of specific software engineering knowledge to non-software domain engineers. The domain engineers use the domain specific model to compose models of instances of a “robotic system product line.” This allows project leaders to control development and manage differences while leveraging common characteristics of the application domain. [13].

2.8. Code Composition / Generation

The ultimate goal of this research is to free the domain engineer from the arduous task of creating code for prototype robotic systems. All code needed should be created by software expert and stored in a repository. The domain engineer selects the icons that represent collections of code. The domain engineer selects a particular code by completing an annotation in the domain model.

The domain model completed by the domain engineer is represented by an XML file. This file contains all the information needed to recreate the domain model. It also contains all the information needed to compose components and create glue code for the robotic system.

Table 1 is a fragment of code from the domain model represented in Figure 5 above. Remember, this is a simplified model created for illustration purposes only, there are no attributes associated with the atoms (icons) or connections (lines). It also does not have lower level components associated with it that would be necessary to completely configure the artifacts for use in a prototyping system.

Even so, complete robotic system code could be created from the XML file represented in Table 1. One atom is shown, the primitive driver control. Both the primitive driver and the waypoint driver are simply controls in the meta model. In this domain model, they are differentiated by the XML <name> element. They are also assigned “IDs” by the GME environment. The IDs are used later by the XML <connection>/<conpoint> element to specify the source or destination points of the connection.

Adding additional attributes to the meta model will allow additional tuning of the generated/composed coded. Enumerated attributes can constrain the domain engineer to a selection that may be a subset of all the components of this type i.e. a particular set of sensors.

To generate the code, the XML tree is parsed and, in this case, large components (containing predefined wrappers, instrumentation and communications) are assembled for each of the artifacts. The components can be configured by using the parsed XML tree as input to a compositional script written in PERL or RUBY. Another possibility is to transfer the XML to a generative environment, such as an ECLIPSE project. [14].

```
…
<atom id="id-0066-0000000c" kind="Control" role="Control" relid="0x15">
     <name>Primative Driver</name>
     <regnode name="PartRegs" status="undefined">
          <value></value>
          <regnode name="Aspect" status="undefined">
                <value></value>
                <regnode name="Position" isopaque="yes">
        <value>765,72</value>
                </regnode>
           </regnode>
      </regnode>
</atom>
<connection id="id-0068-00000001" kind="Message" role="Message" relid="0x5">
     <name>Message</name>
     <connpoint role="src" target="id-0066-00000001"/>
     <connpoint role="dst" target="id-0066-00000004"/>
</connection>
<connection id="id-0068-00000005" kind="Message" role="Message" relid="0x13">
     <name>Message</name>
     <connpoint role="src" target="id-0066-00000001"/>
     <connpoint role="dst" target="id-0066-00000005"/>
</connection>
<connection id="id-0068-00000008" kind="Message" role="Message" relid="0x16">
     <name>Message</name>
     <connpoint role="src" target="id-0066-0000000a"/>
     <connpoint role="dst" target="id-0066-0000000c"/>
</connection>
…
```

Table 1 Fragment of XML Code Generated by Domain Model Instance

3. RELATED WORK

There is considerable research being conducted in Model Driven Design and Model Driven Architecture. The Object Management Group's (OMG) UML2.0 provides increased support. The Generic Modeling Environment from the ISIS center at Vanderbilt University provides a platform for developing Model Driven designs. The embedded systems community has recognized the power of model driven design for developing software product lines for automotive, signal and aerospace applications. The Eclipse Foundation has several projects focusing on Model Driven paradigms.

3.1. UML2.0

The goal of Model Driven Design is to alleviate difficulties created by the low level of abstraction used in creating today's software systems. The OMG Architecture Group has responded to by embracing a vision to expand UML and provide support for all phases of the software lifecycle [15]. UML2.0 is an outcome of this vision.

UML2.0 supports modeling from different viewpoints. Structural, interaction, activity and state viewpoints have some interdependencies, but allow modelers to concentrate on specific concerns.

UML is still in the development phase as a standard. It is a large and complex, making it difficult to grasp in whole. Experience "from the field" is required to refine and mature the standard.

3.2. Chrysler AG

Czarnecki, Bednasch, Unger and Eisenecker report on their experience at Chrysler AG for automotive and satellite applications [16]. They describe their experience with Model Driven Design and Feature Modeling tool support with the GME tool.

Domain specific concepts and features from the problem space are mapped to a set of combinable elementary components in the solution space using configuration knowledge such as, combination restrictions, default settings and dependencies and construction rules. They use a feature model to define the common and variable features of the products along with supplemental information (binding, priorities etc) unique to the product being developed.

The feature model has a root or concept node and child nodes. The child nodes or sets of child nodes are mandatory, optional, alternative or "or" features. The nodes are combined in various ways to produce an instance of a concept. I.e. A car (concept) can have a manual, automatic or CV transmission, but only one transmission. A car may also have a fossil fuel motor, and electric motor or both.

In the referenced work, they present a UML meta model for feature modeling notation using GME. They also show a derived domain specific model, also using GME.

3.3. Embedded System Control Language

Additional work at Vanderbilt University uses the GME tool, along with Mathworks Simulink and Stateflow tools to create the Embedded Control Systems Language (ESQL) to support development of distributed embedded automotive application [17]. ESQL imports the Simulink/Stateflow models into the GME environment. ESQL is a graphical modeling language for with a suite of sublanguages. Sublanguages are provided to support functional modeling, component modeling, hardware topology modeling and deployment mapping.

The ECSL also has a code generation component. The generated artifacts can synthesize the entire application behavior code, or external application behavior code can be linked in.

4. FUTURE WORK

The next step is implementation of the design environment for prototyping a series of robotic systems. Beginning with simple models, robot simulations and very coarse grained components, the simple models presented earlier will be realized. Continuing, the meta models will be refined to include lower level component composition. A set of robotic artifacts (platforms, controls, OCU's etc.) will have their interfaces wrapped to conform to the JAUS standard. A collection of instrumentation components will be created, as well as several different communications components; TCP/IP and serial to begin with.

As we grow more confident with the meta models and domain specific models, additional artifacts such as mission packages and manipulators will be included both in simulation and physically.

5. CONCLUSIONS

Model Driven Design has great potential to extend the software engineers knowledge to domain engineers. It provides a vehicle for software reuse through the focus on predefined software components.

It simplifies the job of the engineer creating the prototype system by allowing him to focus on the task at hand. It also reduces the time and cost required to evaluate a new application or mission

Since the meta model is the root of all the design efforts and all subsequent activities are captured there is tractability to the initial meta design level.

Finally, prototyping with model driven design provides path forward for implementation of final system.

REFERENCES

[1] W. Smuda, P. Muench, G. Gerhart, K. Moore, “Autonomy and Manual Operation in a Small Robotic System for Under-Vehicle Inspections at Security Checkpoints”, SPIE Defense & Security Symposium, Orlando, FL, April 2002.

[2] C.LickTeig, W. Sanders, P. Durlach, J. Lussier, “Measuring Human Performance in Battle Command”, Army AL&T, May-June 2005, pp16-20.

[3] W. Smuda, L. Freiburger, H. Andrusz, J. Overholt, G. Gerhart, D. Gorsich, “Rapid Infusion Of Army Robotics Technology For Force Protection & Homeland Defence” , Army Science Conference, Orlando, FL, Dec 2002.

[4] “Joint Architecture for Unmanned Systems.” www.jauswg.org

[5] D. Schmidt, “Model Driven Engineering”, IEEE Computer, February 2006, pp 25-31.

[6] C. Szyperski, "Component Technology - What, Where, and How?," *icse*, p. 684, 25th International Conference on Software Engineering (ICSE'03), 2003.

[7] C Szyperski, Component Software: Beyond Object Oriented Programming, Boston, MA, Addison-Wesley, 2002.

[8] K. Czarnecki, U. Eisenecker, “Generative Programming: Methods, Tools, and Applications”, Boston, MA, Addison-Wesley, 2000.

[9] E. Gamma, R. Helm, R. Johnson, J.Vlissides, “Design Patterns”, Boston, MA, Addison-Wesley, 1995.

[10] E. Freeman, E. Freeman, “Head First Design Patterns”, Sebastopol, CA, O’Reilly, 2004.

[11] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann, “ Pattern-Oriented Software Architecture” Vol 2, “Patterns for Concurrent and Networked Objects”, West Sussex, England, 2000.

[12] Generic Modeling Environment, http://www.isis.vanderbilt.edu/projects/gme

[13] K. Czarnecki, M. Antkiewicz, C. Hwan, P. Kim, S. Lau, K. Pietroszek, “ Model-Driven Software Product Lines”, OOPSLA ’05, San Diego, CA, October 2005.

[14] Eclipse, http://www.eclipse.org/

[15] Model Driven Architecture (MDA), Document number ormsc/2001-07-01 Architecture Board ORMSC1

July 9, 2001, http://www.omg.org/mda

[16] K. Czarnecki, T. Bednasch, P. Unger, U. Eisenecker, “Generative Programming for Embedded Software: An Industrial Experience Report”, Proceedings ACM SIGPLAN/SIGSOFT Conference, GPCE, Pittsburgh, PA, October 2002.